Operating Systems

18. Remote Procedure Calls

Paul Krzyzanowski

Rutgers University

Spring 2015

Remote Procedure Calls

Problems with the sockets API

The sockets interface forces a read/write mechanism

Programming is often easier with a functional interface

To make distributed computing look more like centralized computing, I/O (read/write) is not the way to go

RPC

1984: Birrell & Nelson
- Mechanism to call procedures on other machines

Remote Procedure Call

RPC is a set of tools and libraries to give the programmer the *illusion* of calling procedures on a remote system.

Regular procedure calls

The code for generating a normal procedure call is generated by the compiler

You write:

x = f(a, “test”, 5);

The compiler parses this and generates code to:

a. Push the value 5 on the stack
b. Push the address of the string “test” on the stack
c. Push the current value of a on the stack
d. Generate a call to the function f

In compiling *f*, the compiler generates code to:

a. Push registers that will be clobbered on the stack to save the values
b. Adjust the stack to make room for local and temporary variables
c. Before a return, unadjust the stack, put the return data in a register, and issue a return instruction

Implementing RPC

No architectural support for remote procedure calls

Simulate it with tools we have (local procedure calls)

Simulation makes RPC a language-level construct

The compiler creates code to send messages to invoke remote functions

instead of an operating system construct

The OS gives us sockets

Implementing RPC

The trick:

Create stub functions
to make it appear to the user that the call is local

On the client

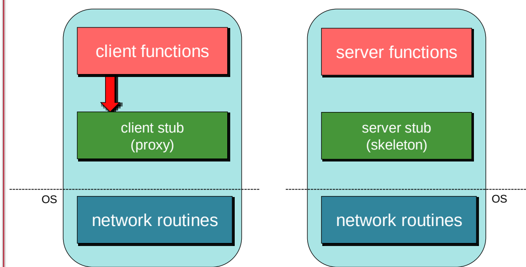
The stub function has the function's interface
Packages parameters and calls the server

On the server

The stub function (skeleton) receives the request and calls the local function

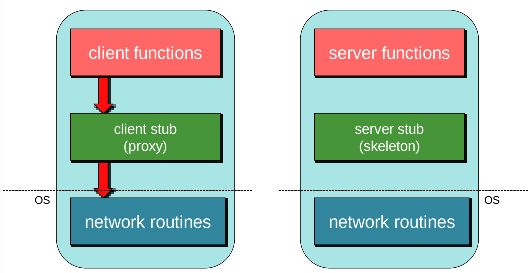
Stub functions

1. Client calls stub (params on stack)



Stub functions

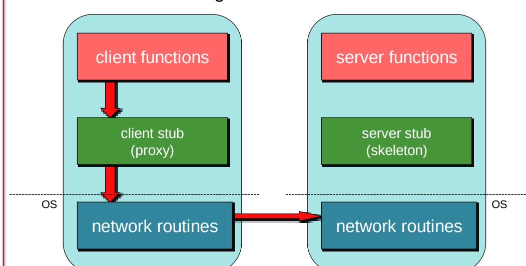
2. Stub marshals params to net message



Marshaling = put data in a form suitable for transmission over a network (serialized)

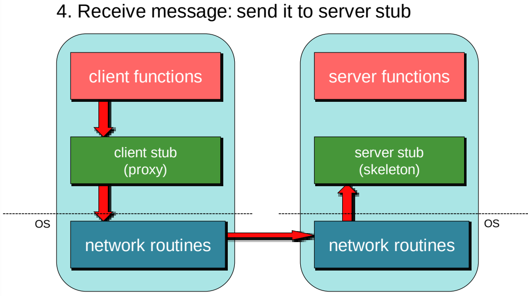
Stub functions

3. Network message sent to server



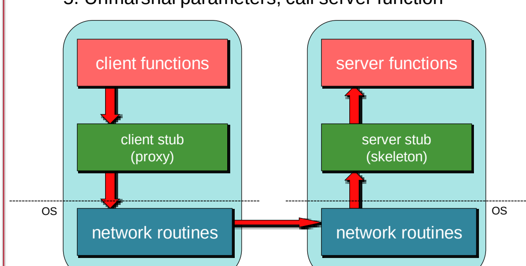
Stub functions

4. Receive message: send it to server stub



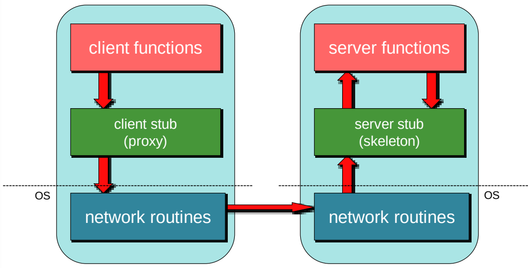
Stub functions

5. Unmarshal parameters, call server function



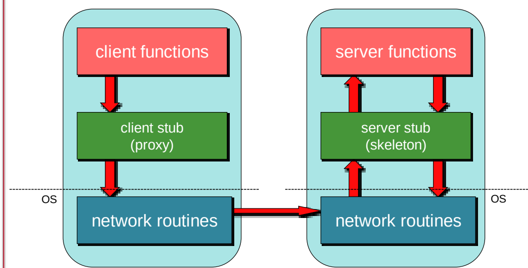
Stub functions

6. Return from server function



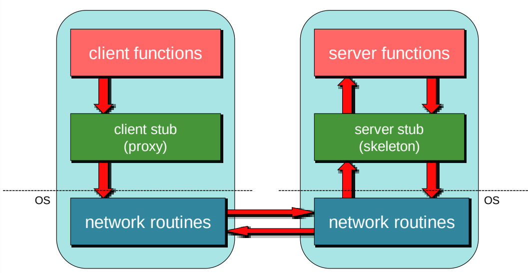
Stub functions

7. Marshal return value and send message



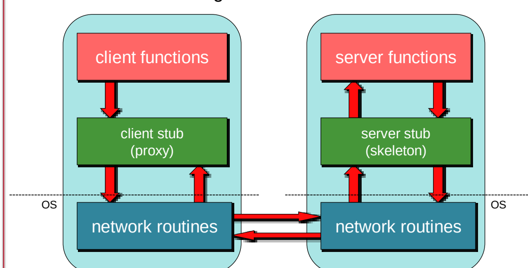
Stub functions

8. Transfer message over network



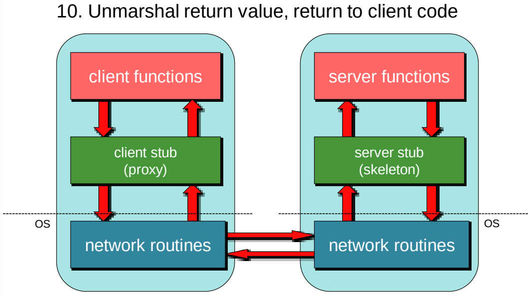
Stub functions

9. Receive message: client stub is receiver



Stub functions

10. Unmarshal return value, return to client code



A proxy looks like the remote function

- Client stub has the same interface as the remote function
- Looks & feels like the remote function to the programmer
  - But its function is to
    - Marshal parameters
    - Send the message
    - Wait for a response from the server
    - Unmarshal the response & return the appropriate data
    - Generate exceptions if problems arise

A skeleton is really two parts

- Dispatcher
  - Receives client requests
  - Identifies appropriate function (method)
- Skeleton
  - Unmarshals parameters
  - Calls the local server procedure
  - Marshals the response & sends it back to the dispatcher
- Invisible to the programmer
  - The programmer doesn't deal with any of this
  - Dispatcher + Skeleton may be integrated
    - Depends on implementation

RPC Benefits

- RPC gives us a procedure call interface
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - Application programmers don't have to worry about details
    - Sockets, port numbers, byte ordering
- Where is RPC in the OSI model?
  - Layer 5: Session layer: Connection management
  - Layer 6: Presentation: Marshaling/data representation
  - Uses the transport layer (4) for communication (TCP or UDP)

RPC has challenges

Parameter passing

Pass by value

- Easy: just copy data to network message

Pass by reference

- Makes no sense without shared memory

Pass by reference?

1. Copy items referenced to message buffer
2. Ship them over
3. Unmarshal data at server
4. Pass *local* pointer to server stub function
5. Send new values back

To support complex structures

- Copy structure into *pointerless* representation
- Transmit
- Reconstruct structure with local pointers on server

Representing data

No such thing as
incompatibility problems on local system

Remote machine may have:

- Different byte ordering
- Different sizes of integers and other types
- Different floating point representations
- Different character sets
- Alignment requirements

Representing data

IP (headers) forced all to use **big endian** byte ordering for 16- and 32-bit values

Big endian: Most significant byte in low memory ← *IP headers use big endian*
– SPARC < V9, Motorola 680x0, older PowerPC

Little endian: Most significant byte in high memory
– Intel/AMD IA-32, x64

Bi-endian: Processor may operate in either mode
– ARM, PowerPC, MIPS, SPARC V9, IA-64 (Intel Itanium)

```
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
            a[0], a[1], a[2], a[3]);
}
```

Output on an Intel:
`44, 33, 22, 11`

Output on a PowerPC:
`11, 22, 33, 44`

Representing data: serialization

Need standard encoding to enable communication between heterogeneous systems

- Serialization
  – Convert data into a pointerless format: *an array of bytes*
- Examples
  – XDR (eXternal Data Representation), used by ONC RPC
  – JSON (JavaScript Object Notation)
  – W3C XML Schema Language
  – ASN.1 (ISO Abstract Syntax Notation)
  – Google Protocol Buffers

Representing data

Implicit typing
– only values are transmitted, not data types or parameter info
– e.g., ONC XDR (RFC 4506)

Explicit typing
– Type is transmitted with each value
– e.g., ISO's ASN.1, XML, protocol buffers, JSON

Where to bind?

Need to locate host and correct server process

Where to bind? – Solution 1

Maintain a centralized DB that can
locate a host that provides a particular service

(Birrell & Nelson's 1984 proposal)

Challenges:
– Who administers this?
– What is the scope of administration?
– What if the same services run on different machines (e.g., file systems)?

Where to bind? – Solution 2

A server on each host maintains a DB of *locally* provided services

Transport protocol

TCP or UDP? Which one should we use?

- Some implementations may offer only one (e.g. TCP)
- Most support several
  - Allow programmer (or end user) to choose at runtime

When things go wrong

- Local procedure calls do not fail
  - If they core dump, entire process dies
- More opportunities for error with RPC
- Transparency breaks here
  - Applications should be prepared to deal with RPC failure

When things go wrong

- Semantics of remote procedure calls
  - Local procedure call: *exactly once*
- A remote procedure call may be called:
  - 0 times:
    server crashed or server process died before executing server code
  - 1 time:
    everything worked well, as expected
  - 1 or more times: excess latency or lost reply from server and client retransmission

RPC semantics

- Most RPC systems will offer either:
  - ***at least once*** semantics
  - or ***at most once*** semantics
- Understand application:
  - **idempotent** functions: may be run any number of times without harm
  - **non-idempotent** functions: those with side-effects
- Try to design your application to be idempotent
  - Not always easy!
  - Store transaction IDs, previous return data, etc.

More issues

Performance

- RPC is slower … a lot slower (why?)

Security

- messages may be visible over network – do we need to hide them?
- Authenticate client?
- Authenticate server?

Programming with RPC

Language support

- Many programming languages have no language-level concept of remote procedure calls
  (C, C++, Java <J2SE 5.0, …)
  - These compilers will not automatically generate client and server stubs
- Some languages have support that enables RPC
  (Java, Python, Haskell, Go, Erlang)
  - But we may need to deal with heterogeneous environments (e.g., Java communicating via XML)

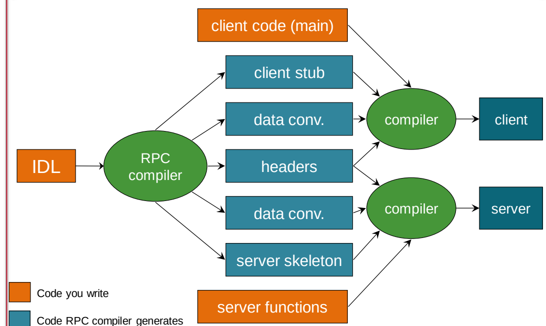
Common solution

- Interface Definition Language (IDL): describes remote procedures
- Separate compiler that generate stubs (pre-compiler)

Interface Definition Language (IDL)

- Allow programmer to specify remote procedure interfaces (names, parameters, return values)
- Pre-compiler can use this to generate client and server stubs
  - Marshaling code
  - Unmarshaling code
  - Network transport routines
  - Conform to defined interface
- An IDL looks similar to function prototypes

RPC compiler



Writing the program

- Client code has to be modified
  - Initialize RPC-related options
    - Identify transport type
    - Locate server/service
  - Handle failure of remote procedure calls
- Server functions
  - Generally need little or no modification

The End